Binary Search: Search method that works on sorted sets of data, typically stored within an array, to find the location matching the search **key**

Basic procedure checks the middle value:

If value matches, return location.

If key is greater than value, we can reduce search space to the upper half

If key is less than value, we can reduce search space to the lower half

Example:

10	11	12	100	200	201	304	500	501	999
0	1	2	3	4 ↑ middle	5	6	7	8	9

key = 12

$12 < 200 \Rightarrow$ value must be located in lower half $(0 \rightarrow 3)$

* Process repeats until key is found, or there are no more elements to search

* Need a method to keep track of the elements we are searching.

↳ can use a left and right index to keep track of current search space

① Init left → 0 and right → size - 1

② while there are elements to search ≡ while left ≤ right

a) middle → (left + right) / 2

b) if data_vals[middle] == key, return middle

c) if data_vals[middle] < key, left → middle + 1

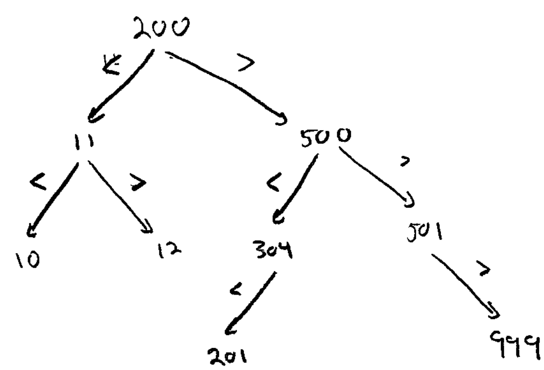
d) else right = middle - 1

③ return -1

* **what is the complexity?**

Search Pattern Example:

10	11	12	200	201	304	500	501	999



* What if we are using some data that is not stored in an array?

↳ Can we create a data structure to store elements/nodes and support the addition and deletion of nodes?

Binary Trees:

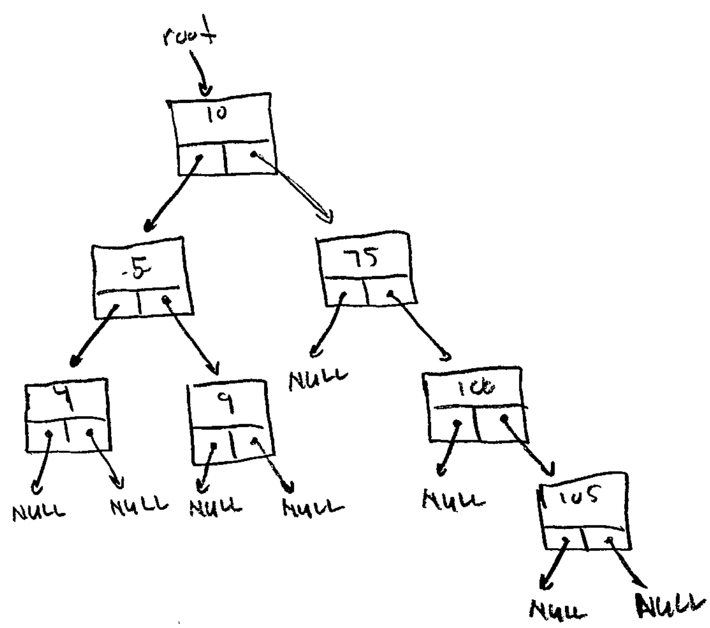
- Hierarchical arrangements of nodes in which each node can have two nodes immediately below it

- Each node consists of data and two pointers to nodes one level below

- Nodes one level below current node are called **children/descendents**
- Node one level above current node is called **parent**
- **Left** and **right** pointers typically used to represent pointers within node

- Node with no children is a **leaf** node
- **Root** node is the single node at the top level of hierarchy

Example:



Note: Nodes could also contain pointer to parent node.

Binary Tree Declarations:

```
typedef struct BiTreeNode_ {
    int data;
    struct BiTreeNode_ * left;
    struct BiTreeNode_ * right;
} BiTreeNode;


typedef struct BiTree_ {
    BiTreeNode * root;
    int size;
} BiTree;


void bitree_init (BiTree *tree);
void bitree_destroy (BiTree *tree);
int  bitree_ins_left (BiTree * tree, BiTreeNode *node, int data);
int  bitree_ins_right (BiTree *tree, BiTreeNode *node, int data);
int  bitree_rem_left (BiTree *tree, BiTreeNode *node);
int  bitree_rem_right (BiTree *tree, BiTreeNode *node);
```

Binary Search Tree: Binary tree in which nodes are organized to aid in effecient searching

- An element within the node is used as a key to determine how nodes are organized

- All elements within the left subtree will have a smaller key than the current node

- All elements within the right subtree will have a larger key than the current node

- Duplicate keys are not allowed.

What is the complexity of a binary search tree search?

Binary Search Tree Interface:

```
int bstree_insert(BSTree *tree, int val);
int bstree_remove(BSTree *tree, int val);
BSTreeNode * bstree_lookup(BSTree *tree, int search_key);
```

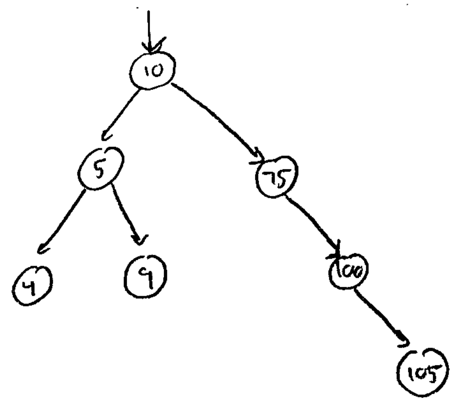
Tree Traversals:

Preorder: Traverse root, Traverse left, Traverse right

Postorder: Traverse left, traverse right, traverse root

Inorder: Traverse left, traverse root, traverse right

Example: Printing Tree



Preorder: 10 5 4 9 75 100 105

Postorder: 4 9 5 105 100 75 10

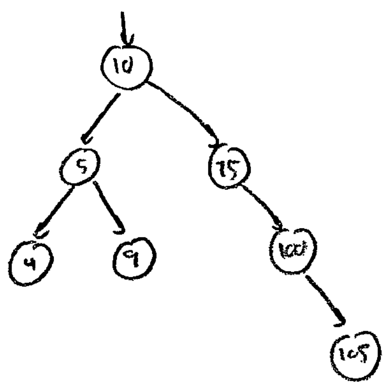
Inorder: 4 5 9 10 75 100 105

Inorder Binary Tree Traversal (Recursion)

```
void bitree_print_inorder(BiTreeNode *node);
```

```
if node != NULL then
    bitree_print_inorder(node->left).
    print node->val
    bitree_print_inorder(node->right)
endif
```

Example:



Note: Integer values below used to illustrate current node

```
bitree_print_inorder (10)
 ↳ if 10 != NULL then
     bitree_print_inorder (10 → left ≡ 5)
      ↳ if 5 != NULL then
          bitree_print_inorder (5 → left ≡ 4)
           ↳ if 4 != NULL then
               bitree_print_inorder (4 → left ≡ NULL)
                ↳ if NULL != NULL then
                  endif
               ←
               print 4
               bitree_print_inorder (4 → right ≡ NULL)
                ↳ if NULL != NULL then
                  endif
               ←
             endif
          ←
          print 5
          bitree_print_inorder (5 → right ≡ 9)
           ↳ if 9 != NULL then
               bitree_print_inorder (9 → left ≡ NULL)
                ↳ if NULL != NULL then
                  end if
               ←
               print 9
               bitree_print_inorder (9 → right ≡ NULL)
                ↳ if NULL != NULL then
                  endif
               ←
             endif
        endif
     ←
     print 10
     bitree_print_inorder (10 → right ≡ 75)
      ↳ if
```